Advanced Parallel Programming: Understanding Locking, Shared Variables and Best Practices

Filip Ekberg
PRINCIPAL CONSULTANT & CEO

@fekberg fekberg.com

Working with Shared Variables

ConcurrentBag<T>

```
var bag = new ConcurrentBag<string>();

Parallel.Invoke(
    () => { bag.Add("Hello");         },
    () => { bag.Add("World");         },
    () => { bag.Add("from");          },
    () => { bag.Add("Pluralsight"); }
);
```

Normal For vs Parallel.For

```
for(int i = 0; i < 100; i++)
{
    // Execute sequentially
}

Parallel.For(0, 100, (i) => {
    // Execute in parallel
});
```

Be careful when **sharing variables** and **resources** in a **parallel process** as you will run into **race conditions**

Always prefer atomic operations over lock when possible as its **less overhead** and performs **faster**

Introducing a Lock

```
Parallel.For(0, 100, (i) => {
    lock(...)
    {

    }
});
```

Introducing a Lock

```
Parallel.For(0, 100, (i) => {
    lock(...)
    {
        // Only one thread at a time please..
    }
});
```

Introducing a Lock

```
Parallel.For(0, 100, (i) => {
    lock(syncRoot)
    {

    }
});
```

If another thread has locked the "door" the thread will be blocked until the "door" is opened

Only **lock** for as **short** of a **time as possible**

Use a **lock** to **access resources** in a **thread-safe** manner

Considerations

- Be careful when adding a lock
- Only lock in short periods of time
- Nested locks can lead to deadlocks
- Do as little work as possible in the lock statement

Next: Performing Atomic Operations

Performing Atomic Operations

Interlocked from System.Threading

“Provides atomic operations for variables that are shared by multiple threads.”

- [Microsoft Docs]

Using Interlocked

```
Interlocked.Increment(ref int thisValue)
Interlocked.Decrement(ref int thisValue)

Interlocked.Add(ref int toBeUpdated, int withValue)
Interlocked.Add(ref long toBeUpdated, long withValue)
```

Incrementing a Variable

```
int result = 0;
Parallel.For(0, 100, (i) => {

});
```

Incrementing a Variable

```
static object syncRoot = new object();

int result = 0;

Parallel.For(0, 100, (i) => {
    lock(syncRoot)
    {
        result += 1;
    }
});
```

Interlocked.Increment

```
int result = 0;

Parallel.For(0, 100, (i) => {
    Interlocked.Increment(ref result)
});
```

Interlocked.Increment

```
int result = 0;

Parallel.For(0, 100, (i) => {
    Interlocked.Increment(ref result)
});
```

Only one thread performs the read & write at a given time. This is faster than using a lock()

Interlocked vs Lock

Interlocked

```csharp
int result = 0;

Parallel.For(0, 100, (i) => {
    Interlocked.Increment(ref result)
});
```

Lock

```csharp
static object syncRoot =
                        new object();

int result = 0;

Parallel.For(0, 100, (i) => {
    lock(syncRoot)
    {
        result += 1;
    }
});
```

Interlocked.Decrement

```
int result = 0;

Parallel.For(0, 100, (i) => {
    Interlocked.Decrement(ref result)
});
```

Interlocked uses **less instructions** and is **faster than** a **lock**

Bitwise Operations with Interlocked

```
int Interlocked.Or(ref int thisValue, int withThis)
int Interlocked.And(ref int thisValue, int withThis)
```

Returns the original value of “thisValue”

Set a New Value

```
T Interlocked.Exchange<T>(ref T source, T newValue)
```

Set a New Value

```
T Interlocked.Exchange<T>(ref T source, T newValue)
```

Returns the "source" value

Interlocked.Exchange

```
int Interlocked.Exchange(ref int source, int newValue)
```

Exercise: Use Interlocked.Exchange as a Gate Keep

```
var gate = 0;

// Try locking the gate
var state = Interlocked.Exchange(ref gate, 1);

if(state == 0)
{
    // Gate was previously locked
}
else
{
    // Gate was already locked

    // Back-off for a while
    // Retry
}
```

Next: Deadlocks with Nested Locks

Deadlocks with Nested Locks

Locked? Thread Will Be Blocked Until Opened

```
Task.Run(() => {
    lock(syncRoot)
    {
        Thread.Sleep(5000);
    }
});


Task.Run(() => {
    lock(syncRoot)
    {

    }
});
```

Locked? Thread Will Be Blocked Until Opened

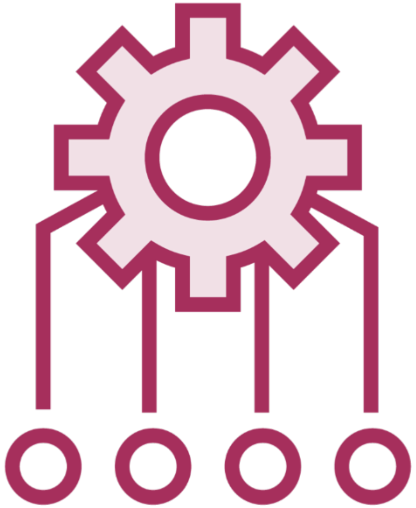
```
Task.Run(() => {
    lock(syncRoot)
    {
        Thread.Sleep(5000);
    }
});


Task.Run(() => {
    lock(syncRoot)
    {

    }
});
```

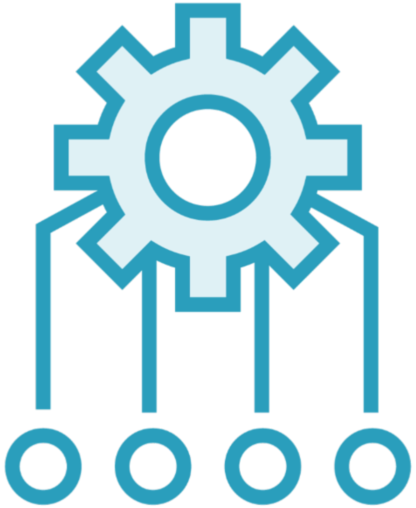
This thread will be locked until "syncRoot" is released (5 seconds)

Deadlock: Fighting over the Same Resource



Thread 1

Waiting for Thread 2 to complete to access **Resource 2**



Thread 2

Waiting for Thread 1 to complete to access **Resource 1**

Avoiding a Deadlock

- Don’t share lock objects for multiple shared resources
- Use one lock object for each shared resource
- Give you lock object a meaningful name
- Don’t use a **string** as a lock
- Don’t use a type instance from **typeof**() as a lock
- Don’t use “**this**” as a lock

Nested locks may result in a **deadlock**

Can You Guarantee That This Doesn't Deadlock?

```
lock(lock1)
{
    Parallel.For(0, 100, (i) => {

        Deadlock();

    });
}
```

Can You Guarantee That This Doesn’t Deadlock?

```
lock(lock1)
{
    Parallel.For(0, 100, (i) => {

        Deadlock();

    });
}

void Deadlock()
{
    lock(lock1)
    {
        Thread.Sleep(1);
    }
}
```

Avoid nested locks and shared locks!

Next: Cancel Parallel Operations

Cancel Parallel Operations

Cancellation

```
CancellationTokenSource cts = new CancellationTokenSource();

var options = new ParallelOptions
{
    CancellationToken =  cts.Token
};

Parallel.For(0, 100, options, _ => {});

Parallel.ForEach(source, options, _ => {});

Parallel.Invoke(options, () => {}, () => {});
```

When a **cancellation is detected**: **No further iterations/operations** will **start**!

Parallel Methods

Built on the Task Parallel Library

Monitors for Cancellation

A task doesn't start when a cancellation token is cancelled

It **won’t stop** executing
already **started operations**

Monitor the Cancellation Token

```
CancellationTokenSource cts = new CancellationTokenSource();

var options = new ParallelOptions
{
    CancellationToken =  cts.Token
};


Parallel.For(0, 100, options, _ => {
    if(cts.Token.IsCancellationRequested)
    {
        // Roll back?
    }
});
```

When to Monitor the Cancellation Token

```
Parallel.For(0, 100, options, _ => {

    FirstExpensiveOperation();

    if(cts.Token.IsCancellationRequested)
    {

    }
    else
    {
        SecondExpensiveOperation();
    }
});
```

Each system executing the code **might complete** it **faster**, or **slower**!

ThreadLocal and AsyncLocal Variables

ThreadLocal<T>

Provides storage that is local to a thread

Note: Task in the Task Parallel Library **reuse threads**!

Thread Local/Static data may be **shared between multiple tasks** that uses the **same thread**

ThreadLocal<T>

```
ThreadLocal<decimal?> data = new ThreadLocal<decimal?>();
```

ThreadLocal<T>

```
ThreadLocal<decimal?> data = new ThreadLocal<decimal?>();

data.Value = 100m;
```

ThreadLocal<T>

```
ThreadLocal<decimal?> data = new ThreadLocal<decimal?>();

data.Value = 100m;

var values = data.Values; // All values for all threads
```

AsyncLocal<T>

"**Represents** ambient **data** that is **local** to a **given asynchronous control flow**, such as an **asynchronous method**."

\- [Microsoft Docs]

When you **write** to **AsyncLocal** a **local copy** will be **created**.

The **outer contexts** value will **not** be **overwritten**!

AsyncLocal<T>

```
AsyncLocal<decimal?> data = new AsyncLocal<decimal?>();


data.Value = 100m;

Task.Run(() => {
    // Local copy created
    data.Value = 200m;
});

// data.Value is stil 100 after the Task.Run!
```

Summary

When sharing variables and resources be cautious with locking

Don’t run expensive operations in a lock

Atomic operations should be preferred when possible as they use less resources, and are faster

Nested locks can cause deadlocks

Don’t share lock objects

Cancelling a parallel operation will prevent further operations from starting

ThreadLocal and AsyncLocal are powerful, but very different from each other